数理解析研究所講究録
586巻 1986年 153-181

An Implementation Scheme for Relational Database Operation Systems based on Demand-Driven Pipeline Processing Concepts

Yasushi KIYOKI[+], Kazuhiko KATO[++] and Takashi MASUDA[+]
清木 康　加藤 和彦　益田 隆司

[+] Institute of Information Sciences and Electronics
University of Tsukuba

[++] Doctral Program in Engineering
University of Tsukuba

ABSTRACT

In this paper, we propose an algorithm which exploits the inherent concurrency between relational operations included in a query. We also present an implementation scheme for a relational operation system based on this algorithm. This relational operation system executes a query concurrently within restricted memory resources by combining pipeline processing with demand-driven control. The main feature of this system is that the highly concurrent pipeline processing is implemented without overflowing the main memory space. We have designed basic primitive operations for the implementation of relational operation systems based on the algorithm and have actually developed a relational operation system by using these primitive operations.

This implementation scheme is appropriate to the implementation under the environment where multiple conventional small computers (workstations) exist on a local-area network. The scheme exploits the concurrency flexibly adapting to an existing system environment.

1. Introduction

To improve the processing performance of relational database operations (relational operations) [1], many processing schemes such as specific algorithms and database machine architectures have been proposed ([2],[3],[4],[5],[6] et al.). Almost all of these schemes try to exploit the inherent parallelism in an individual relational operation. Most database machine architectures are designed on the basis of dedicated hardwares to exploit the parallelism within an individual relational operation.

In this paper, we propose an algorithm which exploit the inherent concurrency between relational operations included in a query. And we also present an implementation scheme for a relational operation system based on this algorithm. This algorithm enables relational operations to be executed concurrently within restricted memory resources by combining the pipeline processing with the demand-driven control. By using this algorithm, a query can be carried out within restricted main memory space without causing memory swapping of intermediate data (intermediate relations). This algorithm is suitable for implementation of a relational database system under the environment where multiple small computers such as workstations exist on the local-area network. In general, in the local-area network environment, the number of computers is restricted, the

capacity of main memory in each computer is also restricted, and every computer does not have a secondary storage. This algorithm makes it possible to flexibly utilize available computers on the local-area network and to exploit the parallelism in such resource-limited environment.

In most existing schemes for executing relational operations, when a query consisting of multiple relational operations is carried out, the granularity of the operand data for each relational operation is a relation. In this case, after the relational operation node completely produces a whole intermediate relation, the subsequent relational operation that consumes the intermediate relation is activated. Therefore, the concurrency of pipeline processing between relational operations is not utilized. Furthermore, memory swapping is necessary between main storage and secondary storage when memory overflow occurs due to intermediate relations. If the granularity transferred between relational operations is set to a page instead of a relation, the inherent concurrency of pipeline processing in a query can be exploited [2]. An execution scheme that implements the pipeline processing by using a data-driven control mechanism had been proposed [3]. However, pipeline processing using data-driven control may cause a memory resource explosion when executing a query. A binary relational operation node is required to compare each outer-relation page with all of the inner-relation pages. Therefore, even if only one inner-relation page and only one outer-relation page remain to be produced, all of the other outer-relation pages and inner-relation pages must be kept in storage. As the result, every node of a query may be forced to keep large amounts of intermediate data in storage. For example, as shown in Fig. 1, if the processing of some node (the Join-1 node) is slow, many intermediate pages are kept in every node.

In the implementation scheme proposed in this paper, a pipeline is constructed among relational operation nodes, and processors are allocated to relational operation nodes on the pipeline. Between nodes on the pipeline, a stream of tuples is passed as an ordered sequence of the intermediate data. As the stream is produced and consumed under the environment of demand-driven control, a query is performed within restricted memory resources. The main features of this scheme are as follows:
(1) Intermediate data does not cause the memory overflow even when the Join operation, the Cartesian-product operation or the Union operation creates a large intermediate relation.

(2) Concurrent pipeline processing is implemented among multiple relational operations in a query.

(3) The response time required to get the first tuple of a query result is very short. A part of the query result can be obtained if only a part of each operand relation exists. This feature is effective for applications which do not require the entire results of a query.

(4) This scheme is suited to the the implementation on multiple small computers connected to a local-area network. This scheme exploits the concurrency flexibly in adapting to an existing system environment.

In order to make the effectiveness of this scheme clear, we show the results of a performance evaluation. These results were obtained by actually processing queries on the proposed relational operation system implemented on a workstation.

2. An Algorithm for Executing Relational Operations

In this section, we explain a basic algorithm for relational operations. This algorithm is based on the concept of demand-driven pipeline processing.

A query such as shown in Fig.1, is executed by this algorithm as shown in Fig.2. When a relational operation node receives a demand from the upper relational operation node, which is the consumer of intermediate tuples produced by the node, the node gets a page from the input buffer and then executes the relational operation until this node completes the production of one resulting page with tuples in the output buffer. The output buffer is then treated as the input buffer for the upper relational operation node. Each relational operation node does not create a whole intermediate relation for a single demand. Each node creates only one page of tuples for a single demand. Each buffer does not require the capacity to store an entire intermediate relation. That is, the buffer size does not relate to the size of an intermediate relation. Each page size is set to a half of the corresponding buffer size, that is, each buffer has the capacity to store two pages. As the size of the buffer allocated to each relational operation node is different, the page size is also different among relational operation nodes, that is, each page size is fixed according to the size of the corresponding buffer.

Immediately after a page is received from the input buffer, the receiving node sends a demand to the lower relational operation node to request the input buffer to be refilled by the following page. In this algorithm, the double buffering mechanism is supported in every buffer. As a result, concurrent pipeline processing is performed between relational operation nodes. By using demand-driven pipeline processing, unary relational

operations (the Selection, the Restriction and the Projection operations), and binary relational operations (the Join, the Union, the Intersection, the Difference and the Cartesian-product operations) can be concurrently executed within the restricted buffer resources without buffer overflow. In particular, in executing the Join, the Union and the Cartesian-product operations, which are the most time-consuming, because these operations produce large intermediate relations, this algorithm shows attractive advantages.

2.1 Unary Relational Operations

A unary relational operation node has one input buffer and one output buffer. Unary relational operations are executed by the following algorithm.

(1) When the unary relational operation node receives a demand from the upper relational operation node, it executes operations (2) and (3) repeatedly until the resulting tuples of one page are created and stored in the output buffer.

(2) A single operand page is read from the input buffer which stores tuples of the operand relation. Then, this input buffer becomes available, and a demand is issued to the lower relational operation node to refill that buffer. As a result, pipeline concurrent processing is implemented between this unary relational operation node and the lower node. (A double buffering mechanism is supported in every buffer.)

(3) The relational operation is executed on the page that has just been read in (2), and the resulting tuples are stored into the output buffer, which corresponds to the input buffer of the

upper node. If the output buffer is filled with resulting tuples, control is returned to (1) and this node waits for the next demand from the upper relational operation node. Otherwise, (2) and (3) are executed repeatedly. If the manipulated page is the last one of the operand relation the execution of this relational operation is terminated.

In the case of the Projection operation node, as it is necessary to eliminate duplicate tuples, this node must not remove the tuples in the output buffer even after those tuples have been read. Therefore, the Projection operation node must reserve the space to store all of the resulting tuples of the Projection operation.

2.2 Binary Relational Operations

In binary relational operation nodes, the operation is carried out by comparing each page of the outer-relation with all of the pages of the inner-relation. Each binary relational operation node has two input buffers and one output buffer. One of the input buffers is used for storing pages of the outer-relation and the other for storing pages of the inner-relation. Binary relational operations are executed by the following algorithm.

(1) When the binary relational operation node receives a demand from the upper node, it executes (2), (3) and (4).

(2) One page of the outer-relation is read from the input-buffer which holds tuples of the outer-relation. Then, a demand is issued to the lower relational operation node to refill that buffer. By issuing the demand before executing the relational

operation, the production of the following operand page in the lower node is overlapped with the execution in this relational operation node.

(3) One page of the inner-relation is read from the input-buffer, and next a demand is issued to the lower relational operation node to refill the vacated input-buffer. As a result, pipeline concurrent processing is implemented between this binary relational operation node and the lower node that constructs inner-relation pages.

(4) The relational operation is executed by comparing the outer-relation page read in (2) with the inner-relation page read in (3). The resulting tuples are stored in the output buffer. When the output-buffer is filled as the result of the demand received in (1), control returns to (1) and execution is stopped until the next demand is issued from the upper relational operation node. Otherwise, (3) and (4) are executed repeatedly. If the compared page of the inner-relation is the last page, the lower relational operation node which creates inner-relation pages is initialized and control returns to (2) in order to compare the following operand page of the outer-relation with all the pages of the inner-relation. (This means recomputation by the lower-relational operation node which creates the inner-relation. The inner-relation is recomputed to compare all the inner-relation pages with each outer-relation page.) If both of the compared pages are last pages, the execution of the relational operation is terminated.

3. An Implementation scheme for Relational Operation Systems

In order to implement the proposed algorithm, a relational operation system must provide the demand-driven control mechanism and the pipeline processing mechanism.

One scheme for the implementation of this algorithm is based on using a dataflow machine with data-driven and demand-driven control mechanisms. In this scheme, the relational operations are described by functional programming language which is suited to the dataflow machine. This scheme realizes demand-driven pipeline processing of relational operations by applying functional programming concepts to relational operations. We have described a relational operation system in the functional programming language Valid [8]. Valid supports the eager and lazy evaluation mechanisms [8],[9],[10] in executing a functional computation [9]. The eager evaluation mechanism and the lazy evaluation mechanism correspond to the pipeline processing mechanism and the demand-driven control mechanism in the dataflow machine, respectively. The execution of relational operations on relations corresponds to functional computation on streams, and the arguments of functions are evaluated by using the eager and lazy evaluation mechanisms. The relational operation system is implemented on a dataflow machine [7] which supports the eager and lazy evaluation mechanisms.

In this paper, we present another implementation scheme for a relational operation system based on the proposed algorithm. This scheme is designed for using conventional computers and a procedural language. This scheme is appropiate to the implementation in an environment where multiple small computers such as workstations exists on a local-area network.

3.1 A single processor environment

In implementing the proposed algorithm in a single

processor environment, a pseudo-pipeline is constructed between relational operation nodes. Each relational operation node acts as a coroutine, and pseudo-pipeline processing is carried out between these coroutines. The query shown in Fig.3(a) is implemented as shown in Fig.3(b). A relational operation node includes an execution management table (emt[]) that specifies the execution environment of a coroutine and indicates the input and output buffers. A buffer management table (bmt[]) is provided in each buffer to indicate the buffer states as shown in Fig.3(b). The demand-driven control mechanism is implemented by activating the suspended coroutine by the execution of a basic primitive operotion "demand". The execution sequence of relational operation nodes, that is the execution sequence of coroutines, is scheduled by a single scheduler. The scheduler passes control to one of the relational operation nodes which receive a demand from an upper node. The choice of the next execution node depends on the scheduling policy. The relational operation node which has obtained control executes a relational operation on an operand page stored in the input buffer. Relational operations are executed by using the basic primitive operations shown in Fig.4. Each node continues the execution of the relational operation until one of the following conditions occurs.

(1) a single output page is created and stored in the output-buffer, or

(2) the processing on the pages stored in the input buffer is terminated.

If the execution is suspended due to (1), the relational operation node stores a "put-wait" in the waiting state of the execution management table, and also saves the resume point of the same node (the relational operation routine). Then, the same node executes the basic primitive operation "acknowledge(sid)" in order to inform the upper node (the consumer) that the output-

buffer (the input-buffer of the upper node) is filled, and returns control to the scheduler. If the execution is suspended due to (2), the node stores a "get wait" in the waiting state of the execution management table, then executes the basic primitive operator "demand(sid)" to the lower relational operation node to request the creation of the following page, and returns control to the scheduler.

3.2 A multiprocessor environment

One or more relational operation nodes are assigned to each processor before starting query processing. The same organization as that in the single processor environment discussed in (1) is implemented in each processor. Therefore, each processor includes its own scheduler, and the scheduler controls the execution sequence of the relational operation nodes allocated to the processor. For each processor, the relational operation nodes and the buffers allocated to the processor are also managed in the same way as that for the single processor environment. Between a relational operation node and another node which are allocated to different processors, the operand data (stream data, that is, tuples) and control signals (demand(sid) and acknowledge(sid)) are transferred through a communication link. The transfer of the "demand(sid)" is shown in Fig.5(a) and is carried out in the following sequence.

1) In processor-1, a relational operation node (Relational Op) issues a basic primitive operation "Demand()" in order to request the following operand page for this relational operation.

2) If the destination node of this demand is allocated to another processor (processor-2), this demand is sent to the communication handler (CP) in processor-1.

3) The communication handler creates a packet including the demand, and then transfers the packet to the communication handler of processor-2.

4) The communication handler of processor-2 accepts the packet and interrupts the execution of the relational operation node (Relational Op) which is currently executing a relational operation with processor-2.

5) In processor-2, the interrupt handler is activated, and the demand issued by the relational operation node in the processor-1 is received by the destination node of the demand. Then, the relational operation node interrupted in 4) is activated, and restarts the execution of the relational operation.

The transfer of an operand page (stream data) and the transfer of an acknowledgement are shown in Fig.5(b) and are carried out in the following sequence.

1) The relational operation node (Relational Op) in the processor-2, which receives a demand from the upper relational operation node, creates a page and stores it in the output buffer (Buff). Then, this node issues an acknowledgement to indicate that the production of the page is completed.

2) If the consuming relational operation node of the page is in another processor (processor-1), the requirement for the transfer of the resulting page and the acknowledgement are sent to the communication handler of processor-2.

3) The communication handler creates a packet including the page and the acknowledgement, and transfers the packet to the

communication handler in processor-1.

4) The communication handler for processor-1 receives the page and the acknowledgement, and stores the page in the buffer (Buf). Then, the communication handler interrupts the execution of the relational operation node which is currently executing a relational operation in processor-1.

5) The interrupt handler is activated in processor-1, and the arrival of the page is acknowledged by a signal to the relational operation node which had issued the demand. Then, the relational operation node interrupted in 4) is activated and restarts the execution of the relational operation.

As mentioned above, the proposed algorithm is easily implemented under the multiprocessor environment by the extension of the relational operation system implemented in the single processor environment.

4. Query Processing

In this section, we discuss the effectiveness of concurrent pipeline processing implemented on the basis of the proposed relational operation algorithm. The typical query shown in Fig.6 is used to evaluate the effectiveness of pipeline processing. This query consists of six Join operation nodes and an output node for the creation of the resulting data. Nodes 0 - 6 are Join nodes and node 0' is the output node. We actually developed a relational operation system on a workstation (Sun-2, Unix4.2BSD [11]) as denoted in section 3.1. Furthermore, we simulated a multiprocessor environment discussed in section 3.2 by this

relational operation system. The query was executed under three processing environments on the developed relational operation system.

The three processing environments (A, B and C) were set as follows :

(The total amount of buffer spaces is equal for each of the three processing environments.)

A : The same amount of buffer space is allocated to each Join node.

B : The query is reconstructed according to the join selectivity factor "jsf" ((the number of tuples in the intermediate relation) = jsf*(the number of tuples in outer-relation)*(the number of tuples in the inner-relation)). The Join nodes that have the highest jsf among the same level nodes in a query (node-1 and node-2 are in the same level, and node-3, 4, 5 and 6 are in the same level) are designated to be the nodes (node-0, node-2 and node-6) to create the inner-relations on pipeline.

C : After the reconstruction of the query as in "B", the buffer space is allocated according to a set condition to utilize the concurrency of pipeline processing. The condition is defined as follows. We consider a query consisting of three Join operations (four-way Joins) as shown in Fig.7. The size of the input-buffer for storing outer-relation pages is set to "bs1" (the unit is the number of tuples) and that for storing inner-relation pages is set to "bs2" in the Join-3 node. If, while the Join-3 node is comparing among currently stored pages in input buffers, the Join-2 node completes the production of the following inner-relation page, the delay of the pipeline is eliminated. That is to say, in the Join-3 node, the suspension of the relational operation execution which occurs due to the absence of the following inner-relation page is the delay of pipeline

processing. In order to eliminate the delay in pipeline processing, the size of the input buffer is set according to the following formula. In this evaluation, it is assumed that each join node executes the operation between two operand pages by using the nested-loop algorithm.

(P2,P3 : the number of processors in the Join-2 node and the Join-3 node respectively,
N11,N12 : the number of tuples in the outer-relation and the inner-relation of the Join-1 node respectively,
N211,N22 : the number of tuples in the outer-relation and the inner-relation of the Join-2 node respectively,
jsf-m : the join selectivity factor of the join node "m" that decides the size of the intermediate relation (jsf*N1*N2).)

1) processing time for the comparison operations in the Join-3 node

(bs1*bs2/p3) : time to compare between two pages
*(jsf1*N11*N12/bs1) : the number of outer-relation pages
*(jsf2*N21*N22/bs2) : the number of inner-relation pages

2) processing time for producing the inner-relation page (bs2)

(bs2/(jsf2*P2)) : time to produce an inner-relation page
*(jsf1*N11*N12/bs1) : the number of recomputation times
*(jsf*N21*N22/bs2) : the number of inner-relation pages

3) the condition for eliminating the delay of the pipeline

1) $\geq$ 2) ---> bs1 $\geq$ P3/(jsf2*P2) ------- formula 3)

If buffer sizes are set by using formula 3), the delay of pipeline processing between every pipeline node in a query is eliminated. Therefore, the overhead for recomputation of a relational operation is eliminated by pipeline processing. If one processor is allocated to each join node (P2 = P3 = 1), formula 3) is defined as bs1 > 1/jsf2. In general, the condition for utilizing the concurrency of pipeline processing between the upper node-m and the node-n that generates the inner-relation of node-m is defined as

bsm $\geq$ 1/jsfn (bsm : the buffer size (the unit is the number of tuples) for the outer-relation of node-m,
jsfn : the join selectivity factor of node-n).

In order to evaluate the effectiveness of the pipeline processing, we have executed a query (Fig.6) on the developed relational operation system. The query is executed under processing environments "A", "B" and "C" shown in Fig.8. We give time charts which indicate the execution state of each relational operation node. The execution time was recorded by the actual query processing on the workstation. Time charts (Fig.9, Fig.10 and Fig.11) show status transitions of each processor in three processing environments (A, B and C) respectively. These time charts were created according to the dependency between the stream generating node and the stream consuming node. These time charts show processor status transitions from the beginning of the query processing for 1000 seconds.

In the proposed algorithm, the line of inner-relation generation nodes becomes the pipeline of the data stream. In Fig.6, the pipeline is built up between nodes 0', 0, 2, and 6. The effectiveness of this relational operation system is remarkable when the throughput of the top node (node 0' or node 0) is high and the output of tuples is continuously performed.

The efficiency of this relational operation system is influenced by the reconstruction of a query and the buffer allocation. By comparing the throughputs (Fig.8) or the time charts (Fig.9, Fig.10 and Fig.11) among three processing environments (A, B and C), it is clear that the effectiveness of pipeline processing is not remarkable in the case of processing environment "A". This is because node 6 becomes a bottleneck, and as a result the throughput of node 0 is very low. On the otherhand, in the case of processing environments "B" and "C", throughputs of the pipeline nodes (node 0', 0, 2, and 6) are high, that is to say, the efficiency of pipeline concurrent processing is remarkable. The effectiveness of pipeline processing in processing environment "C" is especially high.

Therefore, in an environment where join selectivity factors can be estimated, the proposed system shows the best of its pipeline processing advantages. In general, because the precise estimation of join selectivity factors is difficult, it is not easy to create an environment "C". On the other hand, as it is easier to compare join selectivity factors of the Join operation nodes which are in the same level in a query, an environment "B" can be realizied easily. Therefore, in the proposed scheme, if the precise estimation of join selectivity factors is difficult, queries are executed under the execution environment as "B".

5. Conclusions

In this paper, we proposed an algorithm which exploits the inherent concurrency in a query. We also presented an implementation scheme for a relational operation system based on demand-driven pipeline processing concepts. This relational operation system executes a query in a pipeline fashion and within restricted memory resources. The main feature of this system is that highly concurrent pipeline processing is performed without overflowing the main memory space. We have designed basic primitive operations for implementing the proposed algorithm and have implemented a relational operation system by using these primitive operations.

Unlike most of existing relational database machines which try to exploit the parallelism existing within an individual relational operation, this relational operation system exploits the concurrency of pipeline processing between relational operations in a query. This relational operation system is suitable for implementing on a conventional local-area network including multiple workstations.

We are currently developing a relational operation system on the multiple workstations connected to the local-area network. As a result, a database server consisting of multiple workstations will be implemented on our local-area network. We will also consider the allocation problems of memory resources and processor resources.

References

[1] E.F. Codd,"A Relational Model of Data for Large Shared Data Banks," Comm. ACM, Vol.13, No. 6, pp377-397, 1970

[2] J.M. Smith and P. Chang,"Optimizing the Performance of a Relational Algebra Database Interface," Comm. ACM, Vol.18, No.10, pp.568-579, 1975

[3] H. Boral and D.J. DeWitt,"Processor Allocation Strategies for Multiprocessor Database Machines," ACM Trans. on Database Systems, Vol. 6, No. 2, pp.227-256, 1981

[4] S.A. Schuster, H.B. Nguyen, E.A. Ozkarahan and K.C. Smith,"RAP.2 - An Associative Processor for Database and Its Applications," IEEE Trans. on Compt., Vol.c-28, No. 6, pp.446-457, 1979

[5] Y. Kiyoki, Y. Tanaka, N. Kamibayashi and H. Aiso,"Design and Evaluation of Relational Database Machine Employing Advanced Data Structures and Algorithms, Proc. 8th International Symposium on Computer Architecure, pp.407-423, 1981

[6] Y. Kiyoki, M. Isoda, K. Kojima, K. Tanaka, A. Minematsu and H. Aiso,"Performance Analysis for Parallel Processing Schemes of Relational Operations and a Relational Database Machine Architecture with Optimal Scheme Selection Mechanism," Proc. 3rd International Conference on Distributed Computing Systems, pp.196-203, 1982.

[7] M. Amamiya, R. Hasegawa, O. Nakamura and H. Mikami,"A List-processing-oriented data flow machine architecture," Proc. NCC,1982, pp.143-151, 1982

[8] M. Amamiya and R. Hasegawa,"Dataflow Computing and Eager and Lagy Evaluations," New Generation Computing, Vol. 2, No. 2, 1984

[9] P. Henderson,"Functional Programming: Application and Implementation," Prentice-Hall International, Englewood Cliffs, N.J., 1980

[10] K. Pingali and Arvind,"Efficient Demand-Driven Evaluation. Part1," ACM Trans. Programming Languages and Systems, Vol. 7, No. 2, pp.311-333, 1985

[11] Programmers Reference Manual for the Sun Workstation, Sun Micro System, Inc., 1982

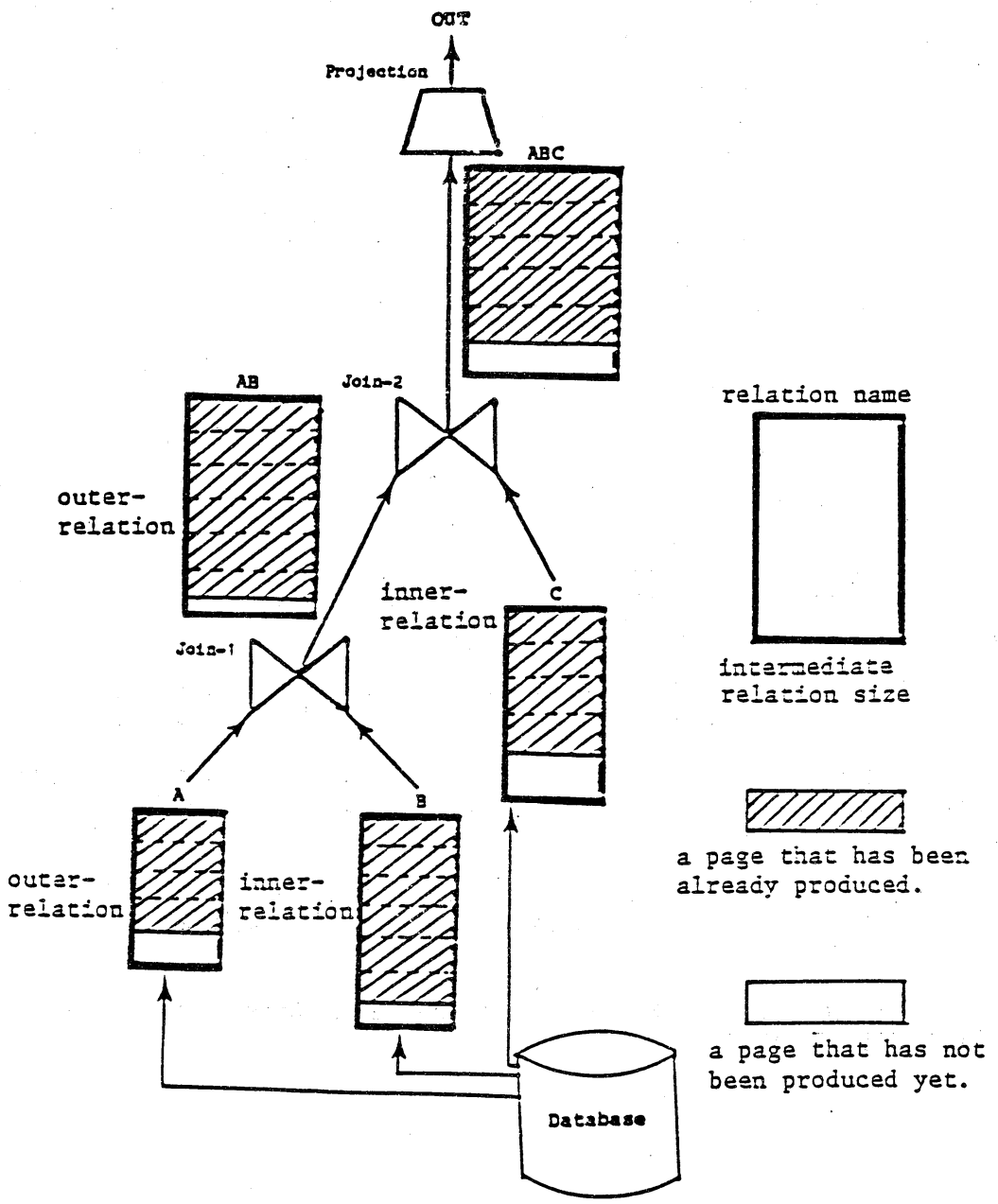

Fig. 1. Query execution in existing schemes.

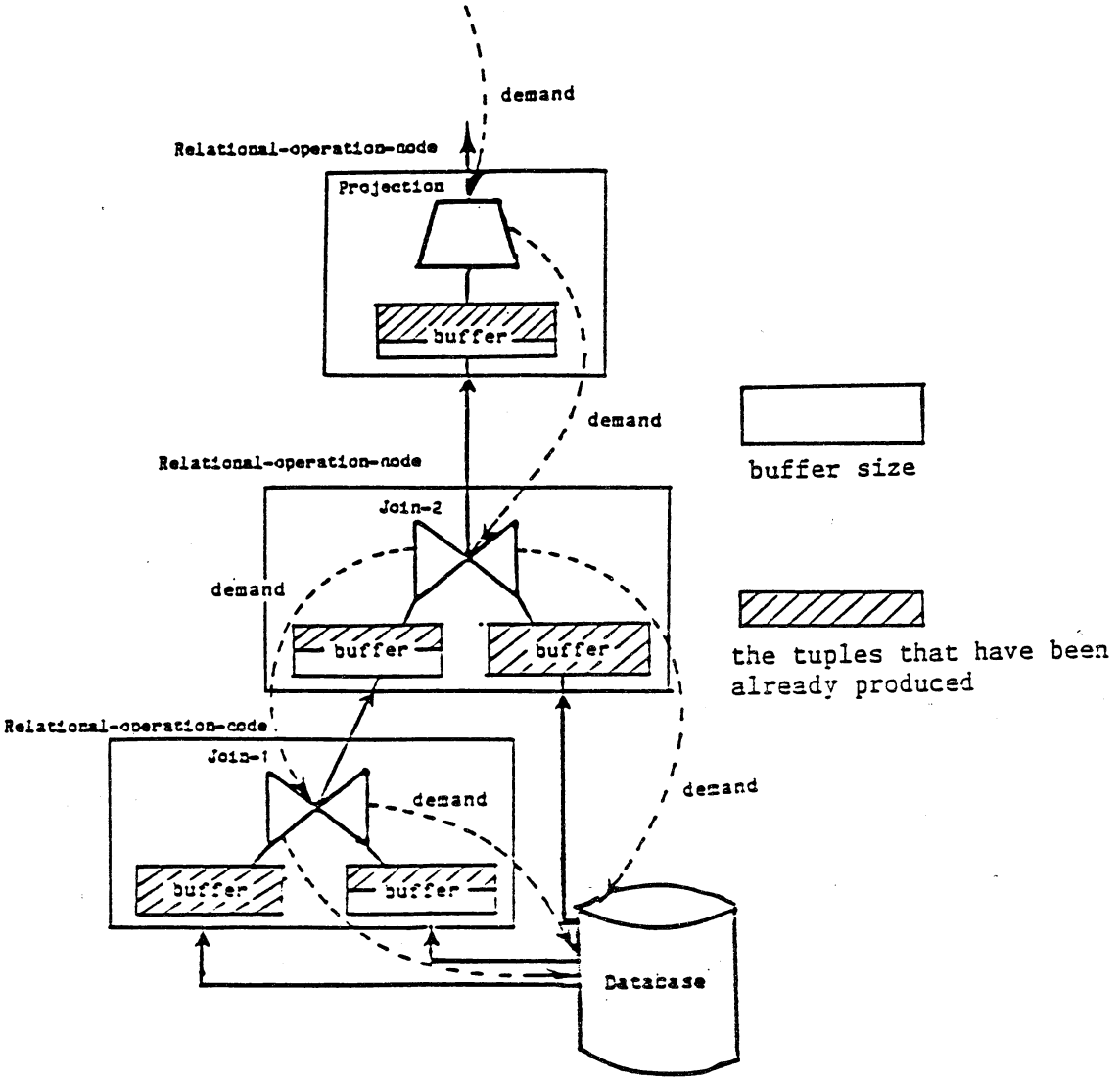

Fig. 2. Demand-driven pipeline processing.

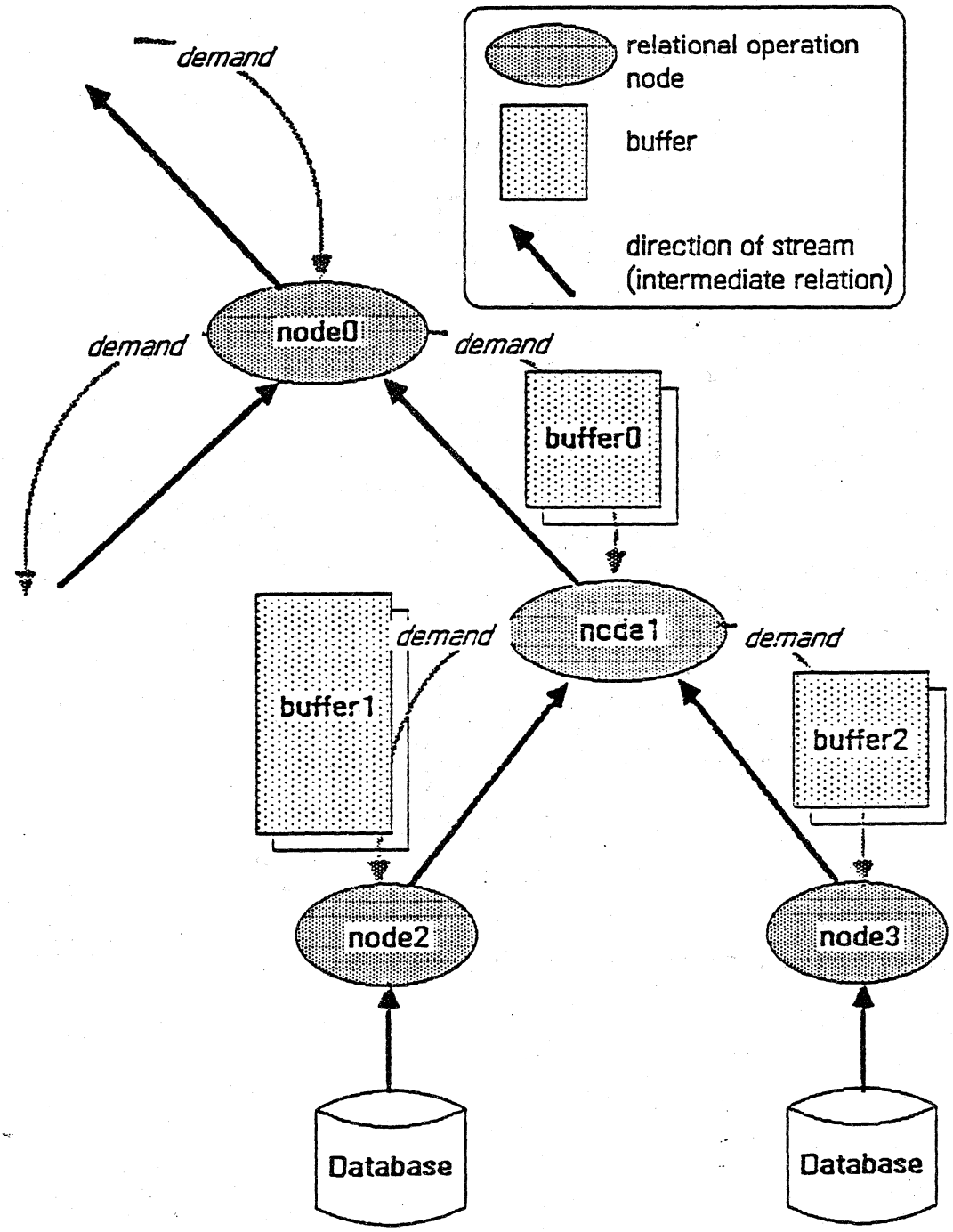
relational operation node
buffer
direction of stream (intermediate relation)
demand
demand
demand
node0
buffer0
node1
demand
buffer1
demand
buffer2
node2
node3
Database
Database

Fig. 3(a). Query.

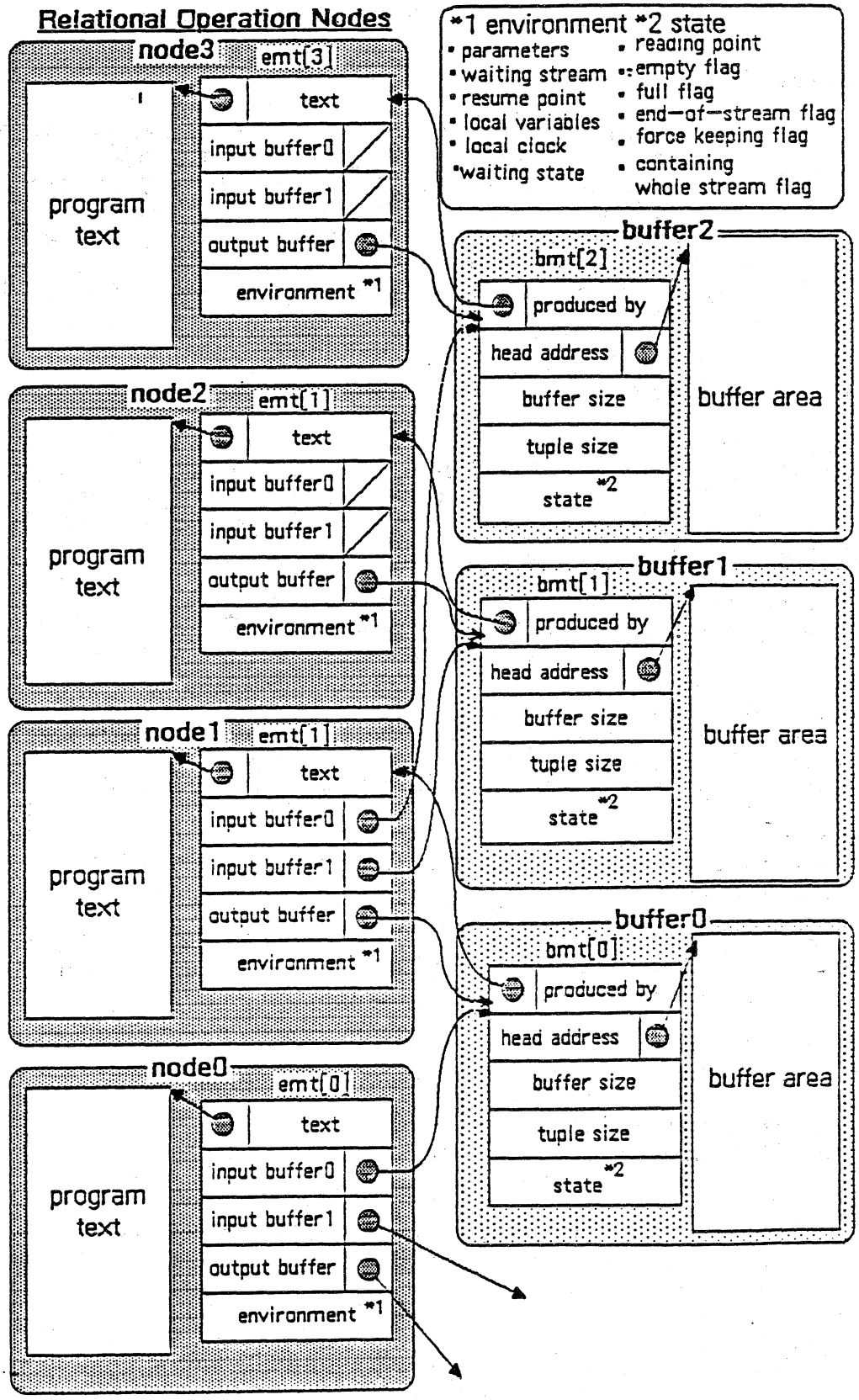

Fig. 3(b). An Implementation scheme for executing relational operation.

Parameters

sid	:	stream identifier (that corresponds to the buffer where a part of the stream is stored),
i	:	i-th tuple in the tuples currently stored in the buffer,
destination	:	work space for storing an operand tuple of a relational operation,
source	:	work space for storing a result tuple of a relational operation.

Primitive Operations

get(sid,i,destination)
moves the i-th tuple of the tuples currently stored in the input buffer to the work space indicated by "destination". The "sid" indicates the identifier of the stream that corresponds to the input buffer.

put(sid,source)
puts a resulting tuple in the work space indicated by source to the stream that corresponds to an output buffer.

rewind stream(sid)
initializes the lower relational operation node indicated by "sid". This operation is used when the recomputation of a stream is necessary in a binary relational operation node.

demand(sid)
requests the creation of the next page for a lower relational operation node.

acknowledge(sid)
indicates the completion of the creation of a page for an upper relational operation node.

check end of stream(sid)
checks the end of the stream.

mark end of stream(sid)
marks the end of the stream.

Fig. 4. Basic primitive operations.

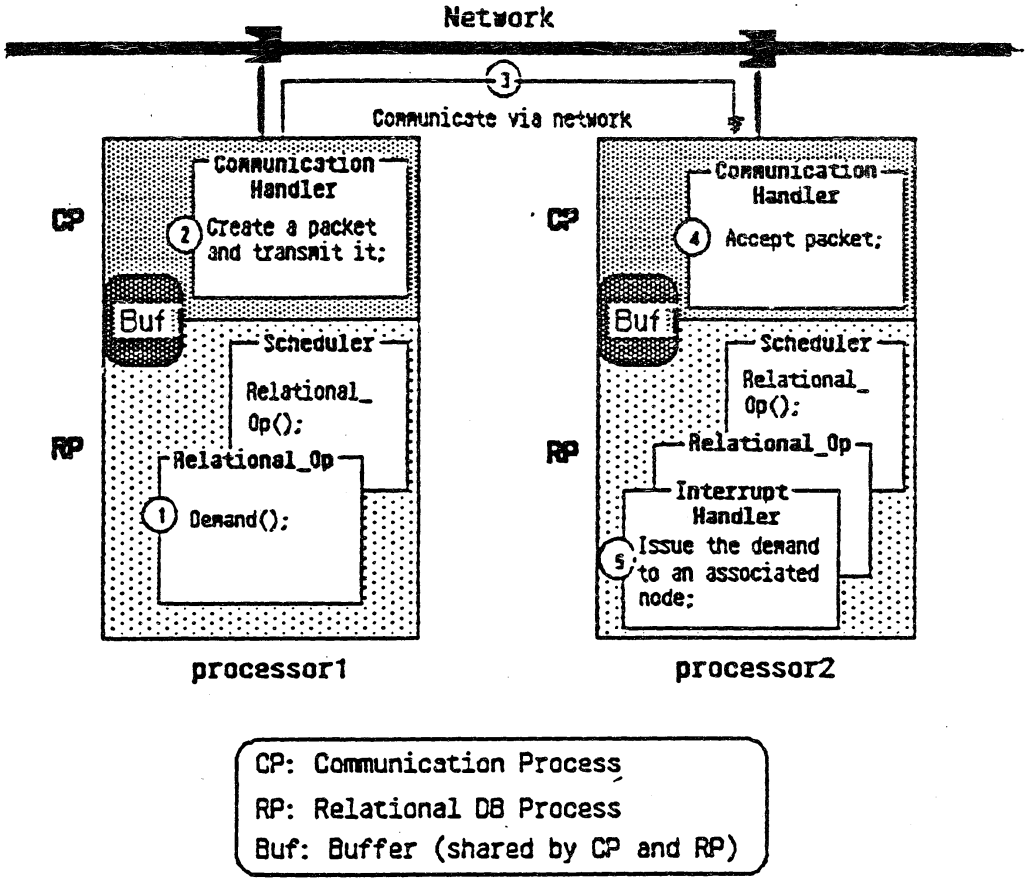

Fig. 5(a). The transfer of a demand.

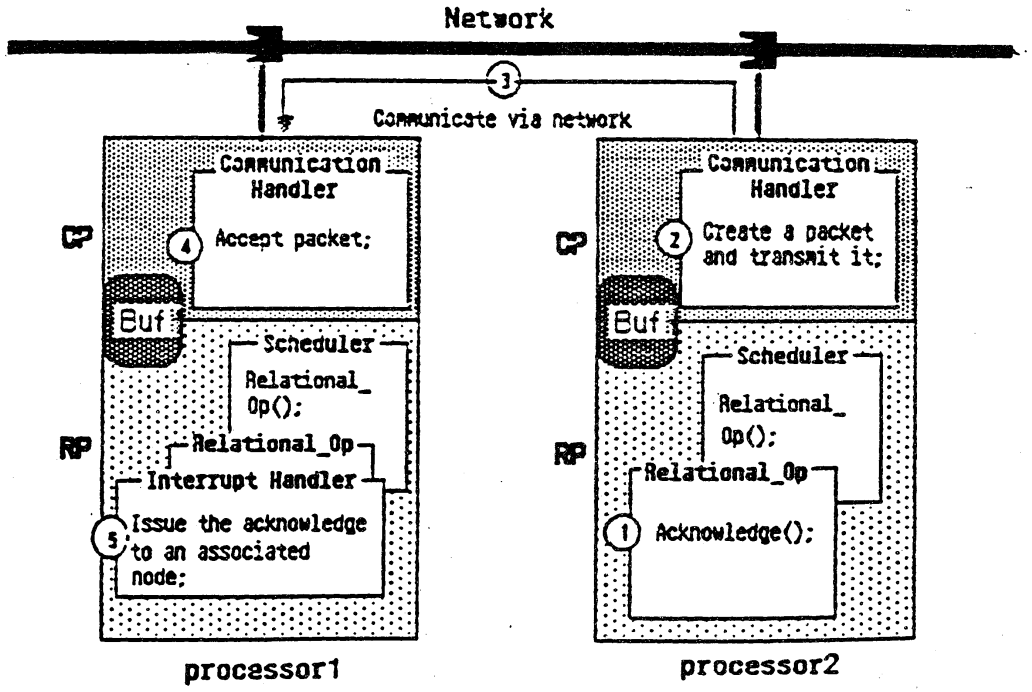

FIg. 5(b). The trsnsfers of a stream and an acknowledgement.

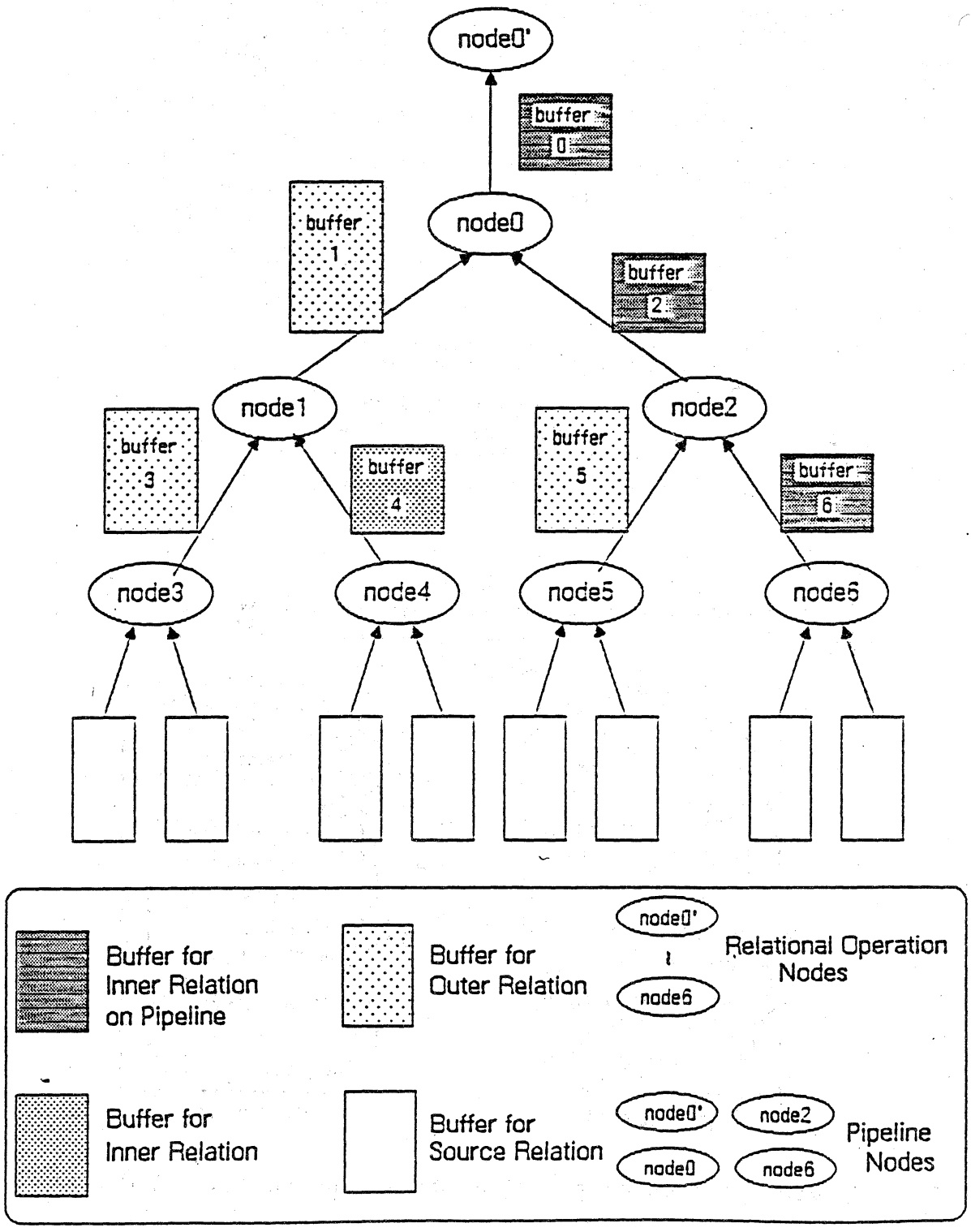

Fig. 6. Example query.

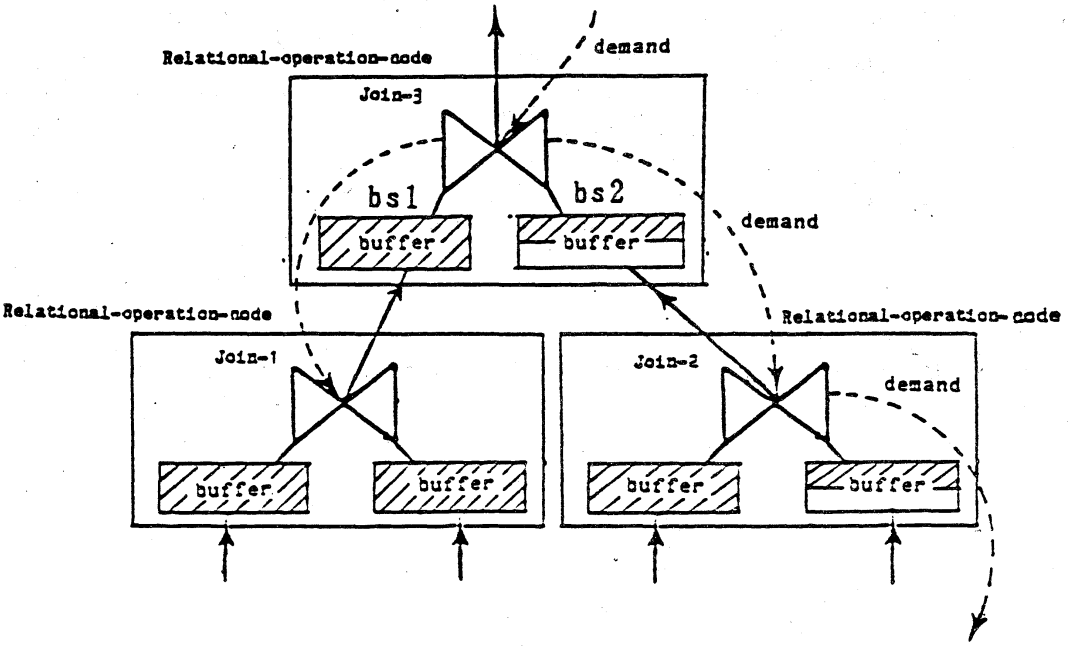

Fig. 7. Pipeline processing.

node/ buffer #	A			B			C		
	j s f	buffer size	through-put	j s f	buffer size	through-put	j s f	buffer size	through-put
0'	--	-	1.20	--	-	24.70	--	-	31.60
0	0.005	10	4.20	0.005	10	60.20	0.005	10	68.40
1	0.01	50	2.50	0.002	50	4.50	0.002	100	9.20
2	0.002	50	32.40	0.01	50	90.30	0.01	30	87.00
3	0.04	50	0.20	0.005	50	2.40	0.005	80	3.60
4	0.02	50	2.80	0.01	50	8.60	0.01	20	10.10
5	0.01	50	12.90	0.02	50	2.70	0.02	30	2.50
6	0.005	50	98.70	0.04	50	55.60	0.04	40	64.40

jsf: Join Selectivity Factor
buffer size: the number of tuples
throughput: (execution time) / (processing time + idle time) * 100 (%),
processing time + idle time = 1000(sec)

Fig. 8. Query processing environments and throughputs.

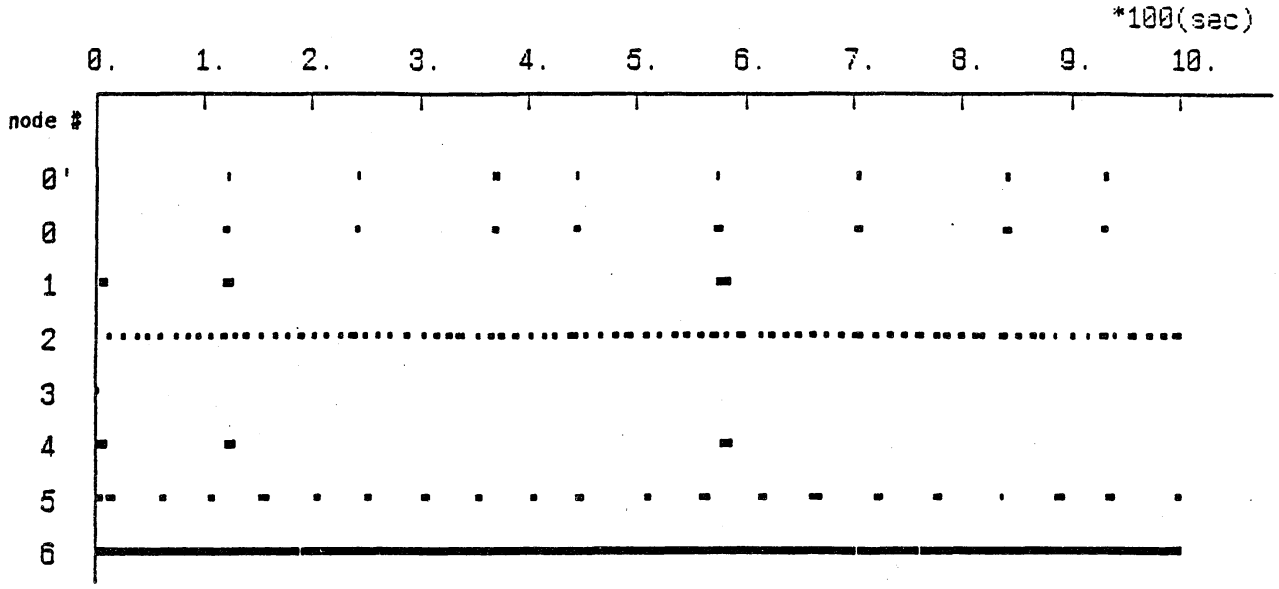

Fig. 9. Time chart for the environment "A".

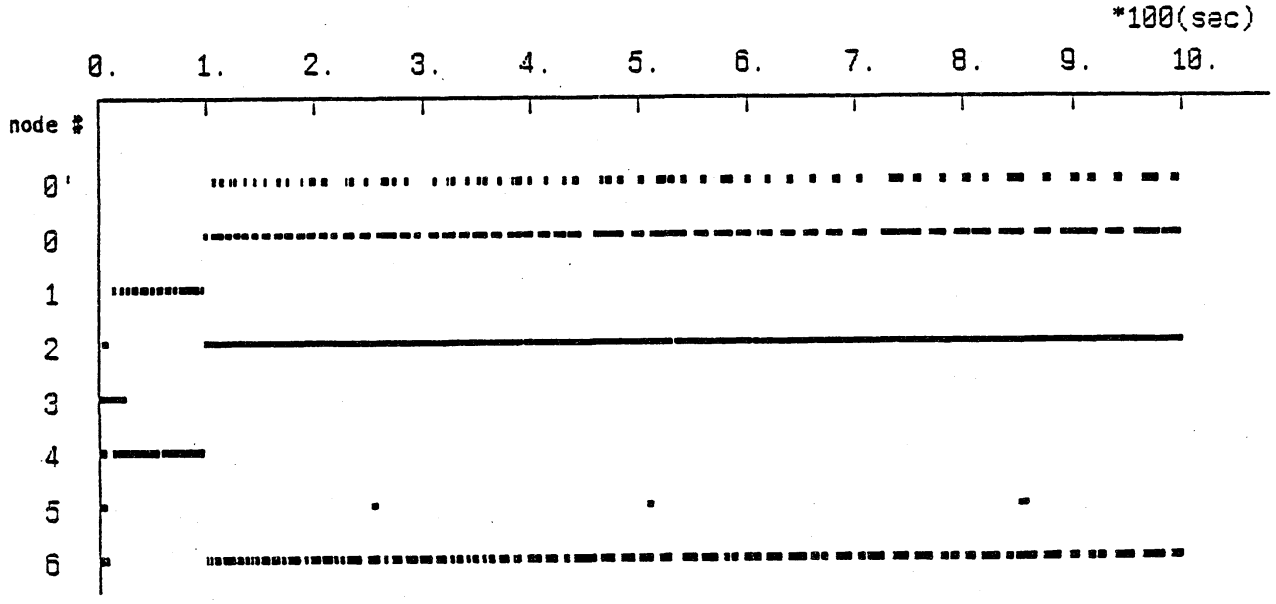

Fig. 10. Time chart for the environment"B".

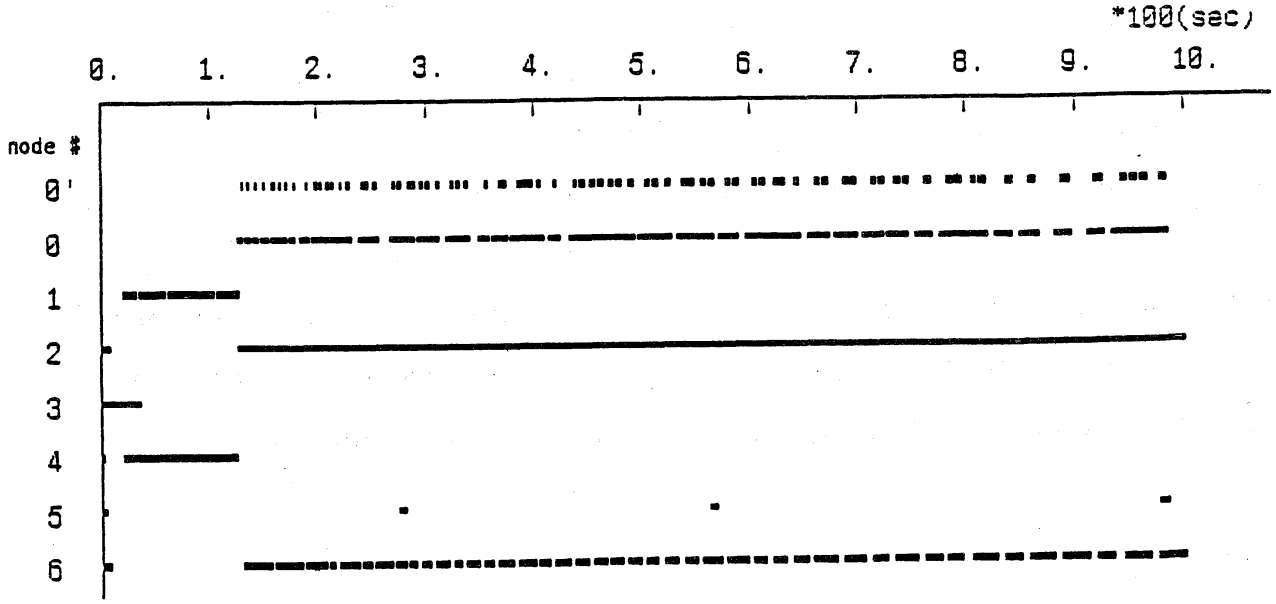

Fig. 11. Time chart for the environment"C".